DOI 10.37882/2223-2966.2026.01.25

# МЕТОДОЛОГИЯ АНАЛИЗА СИСТЕМНЫХ ВЫЗОВОВ В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX СРЕДСТВАМИ STRACE

## METHODOLOGY FOR ANALYZING SYSTEM CALLS IN THE LINUX OPERATING SYSTEM USING STRACE

***I. Otkidach***

*Summary*. This article explores the development of a methodology for analyzing system calls in the Linux operating system using the strace tool to improve the efficiency of software diagnostics and optimization. Particular attention is paid to the systematization of tracing methods and the practical aspects of identifying performance issues at the level of application interaction with the operating system kernel. The goal of the study is to develop a comprehensive methodological approach to system call analysis that ensures effective diagnostics and optimization of industrial information systems. The paper utilizes a classification of tracing techniques, including the basic strace-controlled application launch mode and attachment to running processes via the ptrace mechanism, as well as methods for quantitative analysis of system call metrics and structured troubleshooting. The study resulted in the development of a systematic approach to performance assessment through statistical aggregation, timing analysis, and the identification of inefficient kernel interaction patterns. It also describes an applied troubleshooting methodology with a detailed classification of typical error conditions and blocking operations, accompanied by structured troubleshooting recommendations. The presented methodology ensures increased efficiency of software maintenance processes in industrial information systems through a structured approach to diagnostics at the system call level.

*Keywords*: system calls, strace, Linux system diagnostics, troubleshooting, process tracing, performance analysis, software debugging.

***Откидач Иван Игоревич***
*DevOps-инженер, сертифицированный системный администратор Linux Foundation (LFCS); Сертифицированный администратор Kubernetes (CKA)*
*sre.ivan.public@yandex.ru*

*Аннотация*. Статья посвящена разработке методологии анализа системных вызовов в операционной системе Linux с использованием инструмента strace для повышения эффективности диагностики и оптимизации программного обеспечения. Особое внимание уделяется систематизации методов трассировки и практическим аспектам выявления проблем производительности на уровне взаимодействия приложений с ядром операционной системы. Целью исследования является формирование комплексного методологического подхода к анализу системных вызовов, обеспечивающего эффективную диагностику и оптимизацию промышленных информационных систем. В работе применяется классификация техник трассировки, включающая базовый режим запуска приложений под контролем strace и присоединение к функционирующим процессам посредством механизма ptrace, а также методы количественного анализа метрик системных вызовов и структурированного траблшутинга. В результате исследования разработан систематизированный подход к оценке производительности через статистическую агрегацию, временной анализ и выявление неэффективных паттернов взаимодействия с ядром, а также описана прикладная методология траблшутинга с детальной классификацией типовых ошибочных состояний и блокирующих операций, сопровождающаяся структурированными рекомендациями по устранению неисправностей. Представленная методология обеспечивает повышение эффективности процессов сопровождения программного обеспечения в промышленных информационных системах за счет структурированного подхода к диагностике на уровне системных вызовов.

*Ключевые слова*: системные вызовы, strace, диагностика Linux-систем, траблшутинг, трассировка процессов, анализ производительности, отладка программного обеспечения.

## Введение

Цифровая трансформация предъявляет повышенные требования к методам диагностики и устранения неисправностей на уровне взаимодействия прикладных программ с ядром операционной системы, при этом утилита strace, позволяющая осуществлять трассировку системных вызовов и сигналов и широко применяемая в практике системного администрирования, до настоящего времени не имеет систематизированной методологии использования для решения типовых задач диагностики и оптимизации в научной литературе.

## Материалы и методы

При подготовке настоящей статьи использовались материалы научных публикаций, представленных в базах данных IEEE Xplore, ACM Digital Library и отечественных академических изданиях, охватывающих период с 2015 по 2024 год. Методология исследования основывалась на систематическом анализе литературных источников, посвященных механизмам системных вызовов в операционных системах семейства Linux, инструментальным средствам трассировки процессов и методам диагностики программного обеспечения. Отбор релевантных публикаций осуществлялся по ключевым словам «system calls», «strace», «Linux kernel», «process

tracing», «troubleshooting», после чего проводился критический анализ представленных подходов с целью выявления существующих методологических лакун и формирования комплексного представления о современном состоянии проблематики. Синтез полученных данных позволил структурировать теоретическую базу исследования и обосновать необходимость разработки формализованной методологии анализа системных вызовов средствами strace.

## Литературный обзор

Методология анализа системных вызовов в операционной системе Linux средствами strace базируется на фундаментальных принципах организации операционных систем, изложенных в работах А.В. Батаева [2], и И. Афанасьева [1], где операционная система рассматривается как интерфейс между приложениями и аппаратным обеспечением. Механизмы реализации системных вызовов в ядре Linux детально описаны I. Ajagbe [5], что составляет теоретическую основу для понимания работы утилиты strace. Практические подходы к использованию strace для диагностики производительности и отладки систематизированы в работах M. Boelen [6] и E. Docile [7], которые предложили классификацию техник трассировки и паттернов выявления узких мест в работе приложений. И.А. Стефанова и А.А. Пестов [4] рассмотрели место strace в экосистеме инструментов мониторинга Linux-систем и описали типовые сценарии его применения. Расширенные возможности анализа системных вызовов для задач безопасности представлены в исследованиях T. Nguyen с соавторами [10], K. Kamaluddin [9], а также B.P. Gond и D.P. Mohapatra [8], которые разработали методы обнаружения вредоносного программного обеспечения на основе анализа паттернов системных вызовов. Следует отметить, что в существующей литературе отсутствует комплексная методология, объединяющая техники трассировки, процедуры статистического анализа и алгоритмы диагностики проблем в единую систему.

Научная новизна исследования заключается в описании комплексной прикладной методологии диагностики и устранения неисправностей (траблшутинга) программных систем в среде Linux посредством систематического применения инструментария strace, которая представляет собой структурированную совокупность методов, приемов и алгоритмов анализа системных вызовов, применимых к широкому спектру типовых проблемных ситуаций.

## Результаты

Системные вызовы представляют собой программный интерфейс взаимодействия между пользовательским пространством и ядром операционной системы Linux [5], являясь единственным легитимным механизмом запроса привилегированных операций, таких как работа с файловой системой, управление процессами, сетевое взаимодействие и доступ к аппаратным ресурсам. Утилита strace реализует мониторинг системных вызовов посредством механизма ptrace(2), функционируя как специализированный отладчик, перехватывающий каждый переход через границу пользовательского пространства и ядра и фиксирующий имена вызовов, их аргументы, возвращаемые значения и коды ошибок.

Объектами анализа выступают как успешные вызовы, так и неудачные попытки с кодами ошибок (errno), при этом каждый системный вызов характеризуется набором параметров — дескрипторами файлов, указателями на буферы, флагами доступа и структурами метаданных, семантика которых определяет поведение ядра и позволяет реконструировать логику работы приложения [2]. Классификация системных вызовов в Linux представлена на рисунке 1.

Каждая категория системных вызовов, представленная на рисунке 1, характеризуется специфическими паттернами использования и типичными аномалиями, что формирует основу для целенаправленной диагностики


Рис. 1. Классификация системных вызовов в Linux (составлено автором на основе [1])

с применением фильтрации и агрегации данных трассировки.

Утилита strace предоставляет разнообразные методы трассировки системных вызовов, адаптированные к различным сценариям анализа и диагностики. Базовый метод заключается в запуске целевого приложения под контролем strace посредством команды вида «strace [опции] путь_к_исполняемому_файлу [аргументы]» (например, $ strace -e read cp ~/. bashrc bashrc) — в этом режиме strace становится родительским процессом для трассируемого приложения и перехватывает все его системные вызовы с момента запуска до завершения. Данный подход оптимален для анализа поведения приложения с самого начала его выполнения, включая фазы инициализации, загрузки динамических библиотек и обработки конфигурационных файлов.

Генерируемый утилитой вывод характеризуется значительной протяжённостью, что исключает возможность его исчерпывающего анализа в рамках данного изложения. Целесообразно ограничиться рассмотрением начальной строки выходных данных, которая в результатах работы «strace» структурирована следующим образом:

— идентификатор системного вызова,
— параметры, передаваемые в системный вызов (заключены в круглые скобки),
— результирующее значение, возвращаемое системным вызовом [7].

Инициирующим системным вызовом в выходном потоке является «execve», функциональное назначение которого состоит в запуске программы с определённым набором аргументов. Параметрическая структура функции «execve» включает в качестве первого аргумента — путь к исполняемому файлу, в качестве второго — массив строковых значений, репрезентирующий аргументы, передаваемые программе (конвенционально первый элемент массива идентифицирует наименование самой программы).

Альтернативный метод предполагает присоединение к уже работающему процессу с использованием опции «-p PID», что является особенно ценным при диагностике проблем в долгоработающих сервисах и приложениях, когда перезапуск процесса нежелателен или невозможен. Присоединение к процессу происходит посредством интерфейса ptrace с кодом операции PTRACE_ATTACH, после чего strace начинает получать уведомления о каждом системном вызове. Важно отметить, что присоединение требует соответствующих привилегий — процесс strace должен иметь право трассировать целевой процесс, что определяется политиками безопасности системы, включая Yama LSM и возможности пользователя.

Операция отсоединения трассировщика как правило не оказывает деструктивного воздействия на жизненный цикл контролируемого процесса, который сохраняет операционную активность после завершения сессии трассировки.

Фильтрация системных вызовов представляет собой сильный метод фокусировки анализа на релевантных операциях, где опция «-e trace=» позволяет задать набор отслеживаемых вызовов или категорий, например, «-e trace=open,close,read,write» ограничивает трассировку операциями файлового ввода-вывода, «-e trace=network» фокусируется на сетевых вызовах (socket, connect, bind, accept, send, recv), а «-e trace=process» отслеживает операции управления процессами (fork, exec, wait, exit). Предопределенные категории включают file, ipc, memory, signal и desc (операции с дескрипторами), инверсная фильтрация реализуется через «-e trace=! набор», что позволяет исключить определенные вызовы из трассировки.

Анализ производительности средствами strace основывается на количественных метриках — частоте системных вызовов, их длительности, распределении времени по категориям операций и выявлении аномалий в паттернах обращений к ядру.

Базовый статистический анализ осуществляется посредством опции «— c», которая генерирует агрегированный отчет по завершении трассировки, который включает информацию о проценте времени выполнения (% time), суммарному времени в секундах (seconds), количестве вызовов каждого типа (calls), среднем времени на вызов (usecs/call) и имени системного вызова (syscall). Данная метрика позволяет быстро идентифицировать доминирующие операции в профиле выполнения приложения, например, высокий процент времени в вызовах read/write указывает на интенсивный файловый ввод-вывод, значительная доля futex свидетельствует о накладных расходах на синхронизацию в многопоточных приложениях, а доминирование poll/epoll характерно для событийно-ориентированных серверов.

Детальный временной анализ требует комбинации опций «-T» и «-tt» для получения как абсолютных временных меток [3], так и длительности каждого вызова, позволяя выявить блокирующие операции — системные вызовы с длительностью, существенно превышающей типичные значения, часто указывают на проблемы производительности, например:

— операция open с задержкой в сотни миллисекунд может свидетельствовать о медленной файловой системе, проблемах с сетевыми хранилищами NFS или конкуренции за блокировки на уровне файловой системы
— вызовы connect с аномально высокой длительностью указывают на сетевые задержки, проблемы

с разрешением имен DNS или недоступность удаленного сервиса.

Паттерны системных вызовов служат индикаторами эффективности алгоритмов и архитектурных решений. Множественные последовательные вызовы read с малыми размерами буферов (например, посимвольное чтение) демонстрируют неэффективное использование системного интерфейса, поскольку расходы производительности на переключение контекста при каждом вызове существенно превышают полезную работу, аналогично — частые операции open/close одних и тех же файлов указывают на отсутствие кэширования файловых дескрипторов [6]. Оптимальные паттерны характеризуются использованием буферизации, пакетной обработки и минимизацией количества системных вызовов при сохранении функциональности.

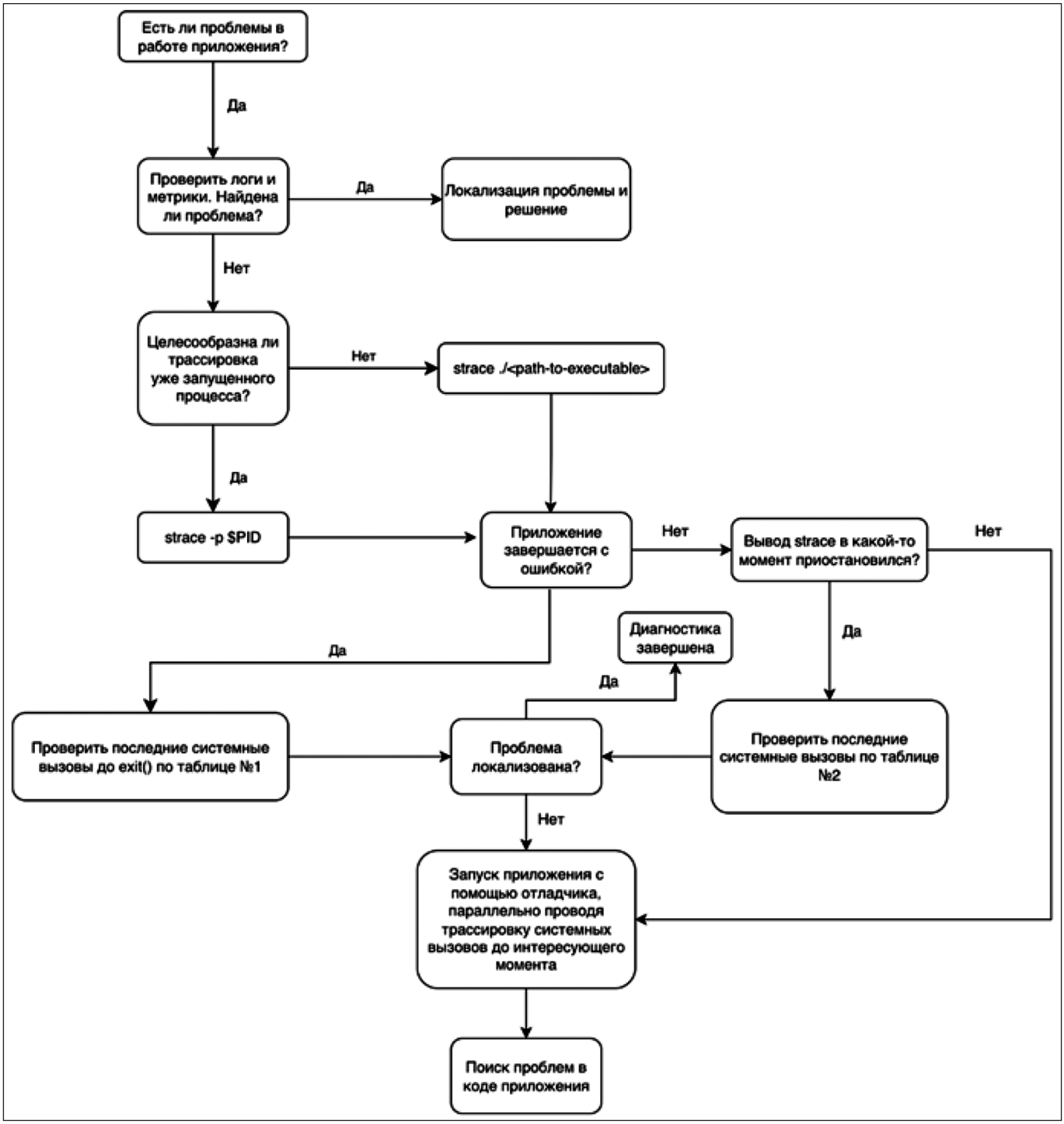


Рис. 2. Общая прикладная методология траблшутинга с помощью strace (разработка автора)

Анализ операций ввода-вывода включает оценку размеров передаваемых данных, выявление частичных операций и неэффективных последовательностей lseek между read/write, сетевые операции анализируются через задержки в accept, неуспешные connect с кодами ошибок и высокую частоту send/recv с малыми объемами данных. Операции управления памятью (mmap, brk) и синхронизация через futex позволяют оценить стратегии аллокации и выявить конкуренцию за блокировки в многопоточных приложениях, профилирование дополняется корреляцией системных вызовов с метриками уровня приложения, а комбинация strace с инструментами perf и eBPF обеспечивает многоуровневый анализ от пользовательского кода до операций ядра.

Диагностика ошибок функционирования приложений методами трассировки системных вызовов представляет собой систематический процесс анализа взаимодействия программы с ядром для выявления аномалий, отклонений от ожидаемого поведения и идентификации первопричин отказов. Методологический подход основывается на интерпретации кодов возврата системных вызовов, где отрицательные значения сигнализируют об ошибке, а конкретный код errno (ENOENT, EACCES, ECONNREFUSED и т.д.) определяет её семантику в соответствии со стандартом POSIX.

Общая прикладная методология траблуштинга с применением strace представлена на блок-схеме (рисунок 2).

Практическая диагностика основывается на классификации ошибок по кодам errno и соответствующим

Таблица 1.

Ситуации завершения приложения с ошибкой

| Пример системного вызова | Проблема | Решение |
|---|---|---|
| openat (AT_FDCWD, «/path/file», O_RDONLY) = -1 ENOENT | Файл не найден | Проверьте путь/опечатки, создайте файл, скорректируйте конфигурацию/переменные окружения. |
| openat (..., «/path/file», O_RDONLY) = -1 EACCES | Нет прав на чтение/директорию | Выдайте права (chmod/chown), проверьте umask, SELinux/AppArmor контексты. |
| stat («/path/dir», ...) = -1 ENOTDIR | Компонент пути не каталог | Исправьте путь, удалите конфликтующий файл и создайте директорию. |
| bind (fd, {.}, ...) = -1 EADDRINUSE | Порт уже занят другим процессом | Найдите владельца порта (ss -ltnp/lsof -i), смените порт или остановите конфликтующий сервис. |
| connect (fd, {...}, ...) = -1 ECONNREFUSED | На целевом порту никто не слушает | Убедитесь, что удалённый сервис запущен/слушает, проверьте firewall/NAT. |
| getaddrinfo («host»,) = -2 | DNS-резолвинг: имя не найдено | Проверьте корректность DNS-записи, настроек DNS или используйте IP. |
| write (fd, ...) = -1 ENOSPC | Нет места на диске | Освободите место, перенесите временные файлы, измените путь вывода. |
| write (fd, ...) = -1 EDQUOT | Превышена дисковая квота | Увеличьте квоту или очистите файлы пользователя/проекта. |
| openat (...,) = -1 EMFILE или = -1 ENFILE | Слишком много открытых файлов (per-proc/системный лимит) | Поднимите ulimit -n/fs.file-max, внедрите пул файловых дескрипторов, исправьте утечки. |
| fork () = -1 ENOMEM или mmap(.) = -1 ENOMEM | Не хватает памяти/ виртуального адресного пространства | Добавьте RAM/swap, снизьте параллелизм, проверьте лимиты RLIMIT_AS/RLIMIT_DATA. |
| execve («/path/app», ...) = -1 EACCES | Файл не исполняемый/ неправильные права | chmod +x, проверьте монтирование с noexec, права и владельца. |
| execve («./prog», ...) = -1 ENOENT (или openat(«/lib64/ld-lin ux.») = -1 ENOENT) | Не найден динамический линкер/ библиотеки | Установите нужные glibc/ld-linux/so-файлы, настройте LD_LIBRARY_PATH/rpath. |
| prlimit64(..., RLIMIT_CORE,) = -1 EPERM / setrlimit(...) = -1 EPERM | Недостаточно прав для смены лимитов | Запустите от нужного пользователя/через sudo, настраивайте лимиты в сервис-юните systemd. |
| mount(.) = -1 EPERM / ptrace(.) = -1 EPERM | Требуются повышенные привилегии/ запрещено политикой | Запустите с root/capabilities (cap_sys_admin и т.п.), скорректируйте SELinux/AppArmor/контейнерные политики. |

системным вызовам. Таблица 1 систематизирует наиболее распространенные ситуации завершения приложения с ошибкой, включая проблемы файлового доступа (ENOENT, EACCES), сетевые конфликты (EADDRINUSE, ECONNREFUSED), ресурсные ограничения (ENOSPC, ENOMEM, EMFILE), проблемы прав доступа (EPERM) и прочие категории неисправностей. Каждая запись содержит пример системного вызова с соответствующим кодом ошибки, её семантическую интерпретацию и рекомендации по устранению.

Таблица 1 содержит дескрипцию наиболее распространённых ошибочных состояний системных вызовов. В ситуации, когда приложение не завершается аварийно, но выходной поток «strace» прекращает обновление, свидетельствуя о приостановке выполнения (зависании), необходимо проанализировать последний зафиксированный системный вызов, который, как правило, находится в состоянии блокировки, ожидая наступления определённого события или доступности ресурса. Для идентификации природы блокировки применяется классификация, представленная в таблице 2.

Если анализ данных, полученных посредством «strace», не позволяет однозначно локализовать источник проблемы, или природа неисправности выходит за рамки системного уровня взаимодействия, методология предполагает переход к более глубокому уровню отладки на котором рекомендуется запуск целевого приложения в среде символьного отладчика (например, GDB) с параллельным ведением трассировки системных вызовов до момента проявления интересующего аномального поведения.

## Обсуждение

Представленные результаты демонстрируют, что утилита strace обеспечивает комплексный механизм анализа взаимодействия приложений с ядром операционной системы Linux на уровне системных вызовов. Профилирование с комбинацией strace с инструментами perf и eBPF обеспечивает многоуровневый анализ от пользовательского кода до операций ядра, что расширяет возможности диагностики за пределы возможностей изолированного применения strace.

Предложенная методология траблшутинга (рисунок 2) представляет собой итеративный процесс последовательного углубления уровня анализа — от высокоуровневых журналов и метрик через системный уровень взаимодействия до внутренней логики приложения. Применение «strace» на промежуточном этапе данной иерархии обеспечивает критически важную информацию о взаимодействии процесса с операционной системой, что в большинстве практических случаев оказывается достаточным для идентификации корневой

Таблица 2.

Идентификация природы блокировки

| Пример системного вызова | Проблема | Решение |
|---|---|---|
| read(0, ... (stdin) — без закрывающей скобки | Ожидает ввод с stdin | Передайте данные |
| accept (fd, ... | Сервер ждёт входящее соединение | Проверьте, что клиент подключается; сгенерируйте трафик для теста; убедитесь, что порт открыт/проброшен. |
| connect (fd, ... зависает) | Блокирующее подключение/ сетевой таймаут | Включите неблокирующий режим + poll/epoll, настройте таймауты сокета, проверьте маршрутизацию/firewall. |
| futex (..., FUTEX_WAIT, ... | Блокировка/ Мьютекс занят, гонка/ deadlock | Соберите backtrace потоков, проверьте порядок захвата локов, включите санитайзеры/трассировку pthread_mutex. |
| epoll_wait (epfd, .../ poll(...) / select(...) | Нет событий на дескрипторах (ожидание) | Убедитесь, что есть источник событий (данные в сокете/pipe, таймеры), проверьте маски/регистрацию дескрипторов. |
| open («/path/fifo», O_RDONLY зависает) | Чтение из FIFO без писателя | Откройте FIFO с писателем (O_WRONLY) или используйте O_ NONBLOCK. |
| waitpid (pid, ... | Ожидание завершения потомка | Проверьте состояние дочернего процесса (strace -f), завис ли он на своём вызове. |
| recvfrom (fd, ... | Ожидает данные из сети | Проверьте, что отправитель шлёт пакеты, настройте таймауты/неблокирующий режим, проверьте MTU/отбрасывание пакетов. |
| nanosleep ({...} / clock_nanosleep | Приложение намеренно «спит» | Проверьте логику ретраев/бекоффов, уменьшите задержки в конфигурации. |

причины неисправности. Если анализ данных, полученных посредством «strace», не позволяет однозначно локализовать источник проблемы, или природа неисправности выходит за рамки системного уровня взаимодействия, методология предполагает переход к более глубокому уровню отладки, на котором рекомендуется запуск целевого приложения в среде символьного отладчика (например, GDB) с параллельным ведением трассировки системных вызовов до момента проявления интересующего аномального поведения.

Предложенный комплексный подход обеспечивает возможность сопоставления состояния пользовательского кода приложения (значения переменных, стек вызовов, точки останова) с последовательностью выполняемых системных операций. Последующий анализ сосредотачивается на логике самого приложения, его внутренних структурах данных и алгоритмах, что позволяет выявить программные дефекты, не проявляющиеся явно на уровне системных вызовов. По достижении успешной локализации и устранения проблемы процесс траблшутинга считается завершённым.

## Заключение

Интеграция методов трассировки системных вызовов в общую стратегию траблшутинга, предполагающую последовательное применение анализа журналов, метрик приложения, strace и символьных отладчиков, обеспечивает многоуровневый подход к решению сложных диагностических задач в производственных Linux-системах, что подтверждает практическую значимость систематизации знаний в данной предметной области для повышения эффективности системного администрирования и разработки надежного программного обеспечения.

ЛИТЕРАТУРА


1. Афанасьев И. Зачем нужна операционная система и как она работает // Основы компьютерных наук. — 2024 [Электронный ресурс] // Режим доступа: https://education.yandex.ru/handbook/vvedenie-v-kompiuternie-nauki/article/zachem-nuzhna-operatsionnaia-sistema (дата обращения: 07.11.2025)
2. Батаев А.В. Операционные системы и среды. — Москва: Академия, 2023. — 288 с. [Электронный ресурс]. Режим доступа: https://academia-moscow.ru (дата обращения: 06.11.2025)
3. Команда strace в Linux // Losst. — 2020. [Электронный ресурс]. Режим доступа: https://losst.pro/komanda-strace-v-linux (дата обращения: 15.11.2025)
4. Стефанова И.А., Пестов А.А.. Обзор утилит мониторинга Linux систем // StudNet. — 2022. — Т. 5, № 6. — С. 5317–5324.
5. Ajagbe I. The Linux Kernel System Call Implementation // Baeldung. — 2023. [Электронный ресурс]. Режим доступа: https://www.baeldung.com/linux/kernel-system-call-implementation (дата обращения: 04.11.2025)
6. Boelen M. Strace cheat sheet // Linux Audit. — 2025. [Электронный ресурс]. Режим доступа: https://linux-audit.com/cheat-sheets/strace/ (дата обращения: 16.11.2025)
7. Docile E. How to trace system calls made by a process with strace on Linux// LinuxConfig. — 2025. [Электронный ресурс]. Режим доступа: https://linuxconfig.org/how-to-trace-system-calls-made-by-a-process-with-strace-on-linux (дата обращения: 14.11.2025)
8. Gond B.P., Mohapatra D.P. System Calls for Malware Detection and Classification: Methodologies and Applications. — 2025 [Электронный ресурс]. Режим доступа: https://arxiv.org/pdf/2506.01412 (дата обращения: 18.11.2025)
9. Kamaluddin K. Dynamic Malware Analysis through System Call Tracing and API Monitoring // ESP International Journal of Advancements in Computational Technology. — 2023. — V. 1, I. 3. — P.167–179.
10. Nguyen T., Orenbach M., Atamli A. Live system call trace reconstruction on Linux // 22nd Annual Digital Forensic Research Workshop Conference USA. — 2022. [Электронный ресурс]. Режим доступа: https://doi.org/10.1016/j.fsidi.2022.301398 (дата обращения: 16.11.2025)



Журнал «Современная наука: актуальные проблемы теории и практики»